NYU CS TR-113
Clark, Dayton

VSH user's guide: a
software environment for

VSH User's Guide:
A Software Environment for Image Processing

by

Dayton Clark
and Robert Hummel

Technical Report No. 113
Robotics Report No. 19

March, 1984

This work has been supported in part by the Office of Naval Research, Grant N00014-82-K-0381, by a grant from the US-Israel Binational Science Foundations, and by grants from Digital Equipment Corporation and the Sloan Foundation.

1 Introduction

Vsh is a UNIX shell that facilitates access to the VICOM image processing system from a host computer. The major functions of *vsh* are:

Issue VICOM commands from the users' terminal on the host computer.

Implement VICOM command files (chain files) with parameters.

Facilitate transfer of images between the VICOM and the host system.

Provide access to the normal UNIX shell facilities.

Provide access to the VICOM from users' programs.

Integrate as much as is feasible the VICOM with other available image processing software and systems.

Vsh is the first stage of the development of an image processing environment. The goal is to develop an environment that is powerful and convenient for the user to use and facilitates integration of new image processing tools.

The implementation described here is for a FOS (Firmware Operating System) VICOM and a VAX 750 running UNIX 4.2bsd. The major task in porting *vsh* to another system would be developing a driver for the image transfer device in the UNIX system.

Vsh provides a command environment in which commands may be processed and executed by the VICOM in an interactive, one-command-at-a-time, mode, or from user programs. The need for a special VICOM shell arises because image processing research frequently requires an extremely high level language which can be easily modified and debugged interactively.

Currently, *vsh* is an interpreted language. Symbol substitution and control features have been kept extremely simple in the interpretive version of *vsh* for efficiency reasons. When greater control structure functionality is needed, *vsh* should be invoked from within a user program. The languages that support invocation of *vsh* are C, FORTRAN 77, and Pascal. A compiled version of *vsh* is being considered. If developed, the compiled version can be expected to support greatly enhanced features.

This document is intended for NYU users of *vsh*, and is periodically revised to account for system changes and new software packages and features. By reading this document, a user will learn how to write the image processing command portions of his or her software. Accordingly, the sections of this document reflect divisions based on intended function, i.e., I/O, control structures, procedure calls, etc. The user should also note that during the early stages of development, this document doubles as the *vsh* specification. For this reason some features of *vsh* do not exist or do not behave as described. Differences between the described and actual system will be found in the associated document "Known Problems in *vsh*" which is included as Appendix D.

2 Overview

The VICOM is an image processing mini-computer which can act as a slave to a host computer, in our case a VAX 11/750. The VICOM acquires, digitizes stores and manipulates images. It acts according to instructions that it receives from the user. The VICOM can obtain instructions from the VICOM CONSOLE, or across a serial line to the VAX. In the normal mode of operation, software operating on the VAX will issue commands to the VICOM across the serial line. The *vsh* program provides a convenient environment for the development of such software.

The VICOM also communicates with the host VAX across a parallel port with direct access to both the VICOM and the VAX memory. This interface is used for image data transfer. *Vsh* manages image transfers between the VICOM and VAX data files.

Figure 2.1 shows the VAX/VICOM configuration. The VICOM user terminals are terminals for the VAX designated for use with the VICOM and *vsh*. Except for this designation they are normal VAX terminals.

Vsh provides an environment on the VAX which allows the user to issue instructions to the VICOM and manipulate images. After a user invokes *vsh* he or she can issue *vsh* instructions interactively or by means of macro files. The user can also execute VAX programs. There are four classes of *vsh* commands:

(1) VICOM commands are instructions which, after symbol substitution, are passed directly to the VICOM, and are interpreted by the VICOM (see the *VICOM Users' Manual*[1]);

(2) Installed *vsh* commands are commands interpreted by *vsh*. These may or may not involve the issuance of commands to the VICOM;

(3) Macro commands are used to invoke a macro file containing *vsh* commands, thereby redirecting the *vsh* command input stream for the duration of the macro file; and

(4) Shell commands are requests for execution of a VAX program and are passed to the host shell.

Vsh can be invoked in two ways. From within the standard shell (e.g. *csh*), *vsh* is simply a program executed in the normal manner. A user can also invoke *vsh* from within programs written in the languages C, FORTRAN, or Pascal. To write image processing software, a user must decide whether the highest level of the software will be written in commands issued to *vsh* (i.e. a *vsh* macro), or in a programming language. The choice will generally depend on the complexity of the control structures required. Ultimately, the choice is not so significant since the two structures for user programs (*vsh* macros calling VAX programs or VAX programs invoking *vsh* commands) recursively pass from one to the other with ease, however the user should give some thought to the basic program structure early in the program design.

The basic operation of *vsh* is:

Get a command line from the input and perform symbol substitution.

Strip the first word (the keyword) from the command and determine the command type.

Perform the appropriate actions on the VICOM and/or VAX.

This is repeated until an **end** or **quit** command or the end of the input file is encountered. To determine the command type, *vsh* matches the keyword against a list of VICOM and installed *vsh* commands. If the match is successful the appropriate action is performed. If no match is found, then *vsh* searches for a macro file whose name matches the keyword (with the extension '.vc'). If a macro file is found then *vsh* accepts commands from that file. *Vsh* first looks in the current user directory for the macro file and then in the global *vsh* macro library. Finally, if no match is found, the command is assumed to be a VAX program or shell command and is passed to the shell for execution.

---

[1]Vicom Systems, Inc., San Jose, California, 1982.

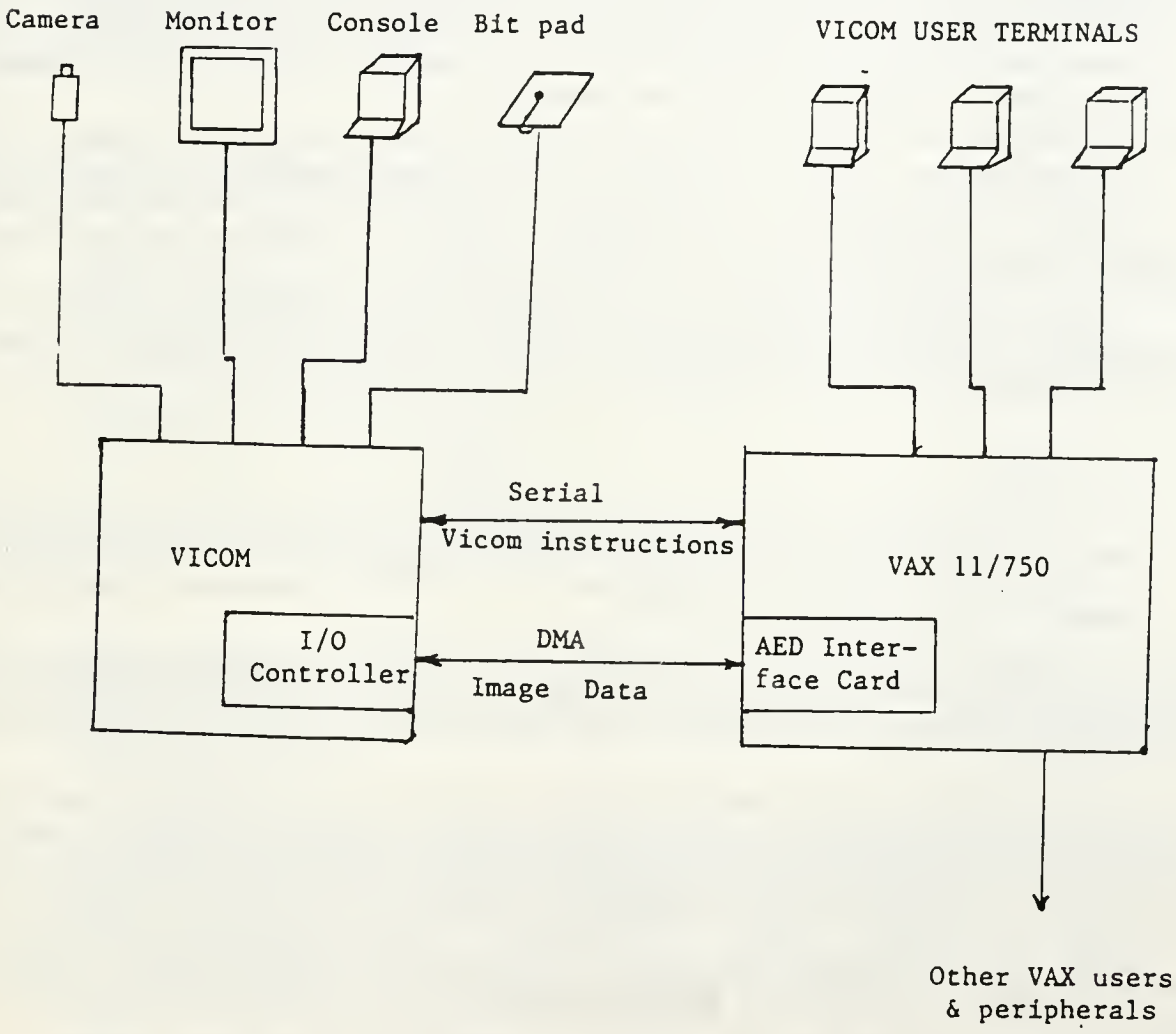

Figure 2.1. VICOM/host configuration.

3 VICOM, *vsh*, and VAX Usage

This section describes the basic procedures for using the VICOM with *vsh*.

3.1 Accounts

In order to use the VICOM and *vsh*, you will need an account on CSD1, one of the computer science department's VAX 11/750's. Accounts will be provided to NYU faculty, staff, and students with valid reasons for using the VICOM. CSD1 uses the UNIX operating system (currently Berkeley UNIX version 4.2). Manuals describing Berkeley UNIX are available in the Courant Institute Library, in the terminal room on the third floor of Warren Weaver Hall, and in the computer vision laboratory on the 12th floor of 715 Broadway. Books on UNIX are available in the University Bookstore and many other bookstores.

You will need to know at least a few of the shell commands, an editor, and probably a programming language – either C, FORTRAN, or Pascal. The shell commands are documented in volume 1 of the UNIX manual. The simplest editor to learn is *ed*, but it's more desirable to use *vi* or *emacs*. Tutorials for *ed* and *vi* are in the UNIX manuals. Many sources are available for learning C, FORTRAN, and Pascal.

You will also need to know the VICOM commands, as described in the *VICOM Users' Manual*. A copy of the manual will be found in the Courant Institute Library and in the computer vision laboratory.

3.2 Power-up

In order to use the VICOM, a user must turn on the equipment and log in at a terminal designated for VICOM users at the computer vision laboratory. Note that if a user simply wishes to edit files or execute programs that do not use *vsh* or the VICOM, any terminal connected to the Academic Computing Facility switch will do.

To power up the system, first turn on the power strip behind the terminals and turn on the terminals. Second, turn on the circuit breaker on the back panel of the VICOM (lower-left) and press the *on* button in the front of the VICOM. Turn on the video monitor with the white switch on its front.

If a camera is needed it must be turned on also. The vidicon camera (the beige and brown camera near the terminals) has a switch at the top. The ccd camera (the red and black camera on the optical table) has a switch on its power supply (a small box connected to the camera). The ccd camera works best if it is allowed to warm up for 30 minutes or so. Only one camera can be connected to the VICOM at a time, you should check to see that the one you wish to use is connected. If you must connect a camera, there are three cables to connect: two sync cables (vertical and horizontal) and a video input. The cables and jacks on the back of the VICOM are labeled.

3.3 Login

Near the VICOM in the computer vision laboratory, there will be one or more VICOM users' terminals, and perhaps a VICOM console. They are marked. To use the VICOM and *vsh* you must login on one of the VICOM users' terminals.

3.4 Invoking *vsh* from the Shell

Before using *vsh*, the VICOM must be ready to accept commands from the VAX. When the VICOM is turned on or reset it expects commands from its console. The prompt

*>

will be displayed on the VICOM console. To have the VICOM accept commands from the host you must press

<esc>-O

on the VICOM's console (that's the escape key and then the capital O). If the VICOM console is not present this step is not necessary. When the VICOM is ready to accept commands from the VAX it will display the prompt

+>

on the VICOM console. The VICOM can be returned to its initial state of expecting commands from the console by again entering

<esc>-O

on the VICOM console. Under various circumstances (eg. the VICOM gets hung or crashes) it may become necessary to perform this step again. In any case the prompt '+>' on the VICOM console indicates that the VICOM is ready to accept commands from the VAX.

Once the VICOM is ready, *vsh* can be invoked either by user programs or from the UNIX shell. The use of *vsh* from the user program is described in Section 3.5. To execute *vsh* from the host shell, enter any of the commands

% vsh
% vsh *p1 ... p9*
% vsh -f *filename p1 ... p9*

Vsh will respond with the prompt

*>

on the VICOM users' terminal. Commands may now be issued to *vsh*. The second and third forms above invoke *vsh* with the parameters *p1 ... p9* at the top level. Parameters are described in Section 5.2. The -f option in the third form directs *vsh* to accept commands from the file *filename* at the top level.

Vsh will not run if you are not at a VICOM users' terminal of if the VICOM is already in use, instead it will give you a message indicating what the problem is. In some situations the parallel connection between the host and the VICOM can not be established, if this is so a message will be printed. In this case the VICOM will still accept commands from the host but images can not be transferred.

Vsh will accept VICOM commands, *vsh* commands and shell commands. VICOM commands can be entered exactly as described in the *VICOM Users' Manual* except that commands **must be entered in lower case**. Other commands are described in the remaining sections of this document.

Figure 3.1 gives a summary of the complete power-up and login procedure.

3.5 Invoking *vsh* from User Programs

Vsh commands can be executed from within users' programs written in C, FORTRAN, or Pascal. Before a program using *vsh* is run the VICOM must be prepared as above, that is, <esc>-O must be entered on the VICOM's console. In addition the *vsh*/VICOM connection must be opened by the following statement:

1. Power strip (and terminals) *on*.
2. VICOM main power *on*.
3. VICOM front panel *on*.
4. Video monitor *on*.
5. Camera *on* and connected, if necessary.
6. <esc>-O on the VICOM console until '+>' is displayed.
7. Login to the VICOM users' terminal.
8. Execute *vsh* or user's program.

Figure 3.1. Power-up and login procedure.

irc = vopen()

Vopen and all (except **verrdesc**) the routines in this section are integer valued functions which return a code number indicating errors or failures during execution. The error code is zero if the routine executes without error. Other error codes are listed in Appendix C.

Note that in the examples given in this section the language is not specified. Except for minor syntactical differences (eg. semi-colons or character case) the routines are invoked in the same manner in C, FORTRAN, and Pascal. Unless the differences are substantial or significant a generic example will be used throughout this document, leaving it to the user to transliterate the

```
/* usevsh.c */
#include <stdio.h>
#include <vsh.h>

main()
{
    int    irc;
    char   command [VCLEN];

    if (irc = vopen())
        printf("%s0, verrdesc(irc));
    else
    {
        while (gets(command) != NULL)
            if (irc = vsh(command)) printf("%s0, verrdesc(irc));
        vclose();
    }
}
```

Figure 3.2. Example use of *vsh* in C.

statement into the appropriate form for the language used.

To invoke a *vsh* command, call the function **vsh** as follows:

irc = vsh(*command*)

where *command* is a character string containing a single *vsh* command. **Vsh** is an integer valued function which returns a code number indicating the errors encountered during the processing of *command*. A series of *vsh* commands can be executed by repeatedly invoking **vsh**, or by passing a macro command to **vsh**.

If the programmer knows that the command to be executed at a certain point of a program will always be a VICOM command then the **vicom** routine can be used. The format is

irc = vicom(*command*)

This will execute considerably faster than the **vsh** routine. It returns an error code as above.

Two additional routines are useful. **Vclose** terminates access to *vsh* and the VICOM. It is a subroutine invoked as follows:

vclose()	in C and Pascal
call vclose	in FORTRAN

Verrdesc is given an integer error code returned by a *vsh* routine and returns a character string containing a description of the error. The purpose is to provide a message that can be shown to the user to indicate the source of an error.

Figures 3.2, 3.3 and 3.4 are examples of programs in C, FORTRAN, and Pascal that use *vsh*, respectively. The programs are equivalent each opens the *vsh*/VICOM connection and then accepts commands and passes them to **vsh** until the end of input is encountered. A description of any **vsh** error is displayed. The constant VCLEN is the limit on size of commands accepted by the VICOM. In the Pascal program the data types 'vstring' and 'vstringpointer' are defined in the include file. The type 'vstring' is an array of characters for holding commands or VICOM error

---

```
      program usevsh
      parameter (VCLEN = 75)
      integer irc
      character*(VCLEN) command
      integer vopen, vsh
      character*(VCLEN) verrdesc

      irc = vopen()
      if (irc .ne. 0) print*, verrdesc(irc)
100   read (5, '(a)', end=900) command
            irc = vsh(command)
            if (irc .ne. 0) print*, verrdesc(irc)
            goto 100
900   continue
      call vclose
      end
```

Figure 3.3. Example use of *vsh* in FORTRAN.

```
program usevsh (input,output);
#include "/a/robotics/h/vshpas.h";
var     irc: integer;
        command: vstringpointer;

procedure getcommand (command: vstringpointer);
var     i: integer;

begin
        i := 1;
        while not eoln do begin
                read (command^ [i]);
                i := i + 1;
        end;
        readln;
        command^ [i] := chr(0); {Necessary terminator.}
end;

procedure puterror(errormess: vstringpointer);
var     i: integer;

begin
          i := 1;
          while errormess^[i] <> chr(0) do begin
              write(errormess^[i]);
              i := i + 1;
        end;
        writeln;
end;

begin
        irc := vopen;
        if irc <> 0 then puterror(verrdesc(irc))
        else begin
                new (command);
                while not eof do begin
                        getcommand(command);
                        irc := vsh (command);
                        if irc <> 0 then puterror(verrdesc(irc));
                end;
                vclose;
        end;
end.
```

Figure 3.4. Example use of *vsh* in Pascal.

messages and 'vstringpointer' is a pointer to a variable of type 'vstring'.

The various *vsh* and VICOM routines described in this document are available in the *vsh* library. To access the library include the '-lv' option in the compile or load statement for the program. The following examples illustrate this.

```
% cc vis.c -lv
% f77 usevsh.f -lv
% pc usevsh.p -lv
```

4 VICOM Commands

VICOM commands will be accepted by *vsh*. These commands are described in the *VICOM Users' Manual*. The command keyword (first word) must be in lower case. VICOM commands are listed alphabetically in Appendix A, and grouped according to functional groups in Appendix B.

Some features and bugs in VICOM commands that are not adequately presented in the manual will be discussed in the following sections.

4.1 Constants

Many commands accept constants as parameters. Typically constants are specified in decimal form. For example

0.5
−.231
−0.12

Decimal constants are always in the range -1.0 to $1-2^{-15}$. Alternatively, constants can be expressed by four hexadecimal digits, followed by the letter capital 'H'. Examples are

8000H (equivalent to -1.0)
C3F2H
4000H (equivalent to 0.5)
7FFFH (equivalent to $1-2^{-15}$)

4.2 Pixel Size

A fully populated image has 16 bits per pixel, 12 for the image and 4 for graphics. For the most part, the arithmetic commands use only the 12 image bits but some operate on the entire 16 bit words. The **adk** (add constant) operates on 16 bit quantities. The manual implies that the shift commands, **lsh** an **ash**, work on the full 16 bits, this is not true. It is apparently impossible to achieve 16 bit arithmetic to any significant degree.

4.3 The cam Command

The **cam** command allows the user to view in real-time the camera input to the VICOM. Its primary purpose is to allow the user to prepare a scene before it is captured via the **dig** command. It is not necessary to invoke **cam** before capturing an image if the scene is already prepared. While the **cam** command is in effect the user should avoid performing unnecessary commands – in particular commands using the cursor or bitpad. Too much activity while the **cam** command is in effect can have undesirable affects including causing the VICOM to crash.

4.4 Acquiring Images

The VICOM can digitize a video signal to obtain a 512 by 512 by eight bit image. Generally, a macro will be used for this purpose (see Figure 5.2). However, for completeness, we describe some of the technical issues in this section.

The camera must be turned on and connected to the VICOM. The vidicon camera has a lens cover which should be placed on the camera when it is not in active use. A scene can be burned permanently into the vidicon tube if it is left looking at the same scene for too long. This is true even if the camera is not turned on.

To view a scene from the camera in real-time the following commands should be issued:

*> int
*> dis *A*
*> cam *A*

A signifys one of the VICOM's images (*A* = 1, 2, ...). The **int** loads the VICOM's display tables so that the appears correct during the **cam** command. The **dis** command causes the indicated image to be displayed and the **cam** command causes the video signal from the camera to be continuously digitized and placed in the indicated image. The scene is now being displayed in real-time, the user can adjust the camera or scene as desired. These first three steps in acquiring an image are solely for the user's convenience and do not effect the acquired image. While the **cam** command is in effect the user should avoid extensive use of other VICOM commands, particularly cursor and bitpad commands. Simultaneous use of the cursor and the **cam** command can cause the VICOM to behave erratically and sometimes crash.

To actually capture an image issue the command

*> dig *A*

where *A* is an image. This places the image data in the high order 8 bits of the indicated image.

Interpreting the 8 bits of captured data as an unsigned integer, we have 0 indicating the lowest intensity and 255 as the maximum. The VICOM processors interpret the pixel data differently. They interpret the high-order twelve bits of the pixel as a two's complement fraction, that is as a signed number in the range $-1.0 \leq x < 1.0$. If we interpret the captured data this way, an intensity value of 0 corresponds to an interpreted value of 0, a midrange intensity of 127 corresponds to the maximum two's complement fraction of almost 1.0, the next intensity corresponds to -1.0, and the highest intensity value of 255 corresponds to almost zero ($-1/128$). Obviously, we want to convert from the input representation to the computational representation. This is done with the following command:

*> two *A* > *A*

which converts the input intensities to the range 0 to 255/256 in the two's complement form. No precision is lost in this conversion since the VICOM actually has 12 bits for the data in each pixel. At this point the display tables should be reset to their original state. To do this issue the command

*> res

Figure 4.1 shows the steps to capture an image.

```
*> int
*> dis A
*> cam A        At this point set the scene.
*> dig A
*> two A > A
*> res
```

Figure 4.1. Step to digitalize image.

Two further notes are worth mentioning. One is that image acquisition works only with full size (512 by 512) images. The system will not object to different memory configurations but the acquired image will always be 512 by 512 pixels. The second is that it is possible to get more precision in the acquired image by averaging several images together.

4.5 The Value 1.0

The two's complement interpretation of pixel values used by the VICOM has no value exactly equal to 1.0. The nearest value is $1-2^{-11}$ or 2047/2048. This is true in pixel values, convolution masks, and arithmetic or logical constants. For instance, the **muk** (multiply by constant) with constant of 1.0 actually causes a slight decay of the image. For occasional usage this is not too serious, however, in iterative procedures this can be a disaster. A possible solution is to change the procedure to use -1.0 instead of 1.0 since the number -1.0 exists in the two's complement interpretation.

4.6 Using the Cursor

The bitpad can be used to control the VICOM's cursor. Two steps must be performed before the bitpad can be used, they are:

```
•> dev (1)
•> wrp
```

The first enables input from the bitpad, the second write protects the graphics nibble of the image planes. These steps need only be performed once after the VICOM is turned on or reset. Now the command

```
•> cur A
```

will display the cursor in image *A*. Press the **stream** button the bitpad and the cursor will follow the bitpad's mouse. Press the **switched stream** button and the cursor will follow the mouse when the mouse's button is pressed. Press the **point** button and the cursor will jump to the current position of the mouse when the mouse's button is pressed. The cursor normally is in the red graphics plane (this can be changed, see the **cur** command in the VICOM manual) and can be moved about without changing the image data, if however, there graphics data in the red plane then the cursor will alter the data as it passes near a pixel.

When commands that use the cursor such as **tra** (for trace) or **pcr** (for print cursor) are executed *vsh* stops accepting input from the user's terminal, it expects input from the bitpad. Place the bitpad in **stream** mode and the cursor will follow the mouse. Press the mouse's button and move the mouse and the appropriate action will be done (i.e. tracing or printing). To exit the command, put the bitpad in **point** mode and press the button on the mouse (it may take several presses), *vsh* should respond with its prompt.

The bitpad demands a lot attention from the VICOM's central processor. For this reason it is wise to "quiet the bitpad" when it's not in use. To do this first press the **point** or **reset** button on the bitpad and or move the mouse off the bitpad working area. If you are done using the bitpad for the session the following commands remove the cursor and disable the bitpad:

```
•> cur A (1,1,0)
•> dev (0)
```

where *A* is the image containing the cursor.

5 Macro Files

One way to write software for the VICOM is to create macro files. A macro file contains a sequence of commands similar to scripts or commands files. Parameters can be passed to macro files, and will be substituted in the commands. Very simple control features are provided, but are kept extremely simple for efficiency reasons. A macro can contain any sequence of commands which can be interpreted by *vsh*.

5.1 Invoking Macro Files

Macro files can be invoked in two ways:

•> *filename p1 ... p9*
•> cha *filename p1 ... p9*

The two forms are essentially identical. Here *filename* is the path to a file named *filename*.vc. The symbols *p1 ... p9* stand for zero or more argument strings which will be substituted for corresponding parameter symbols in the macro file (see below). Macro files are sometimes called "chain files" for historical reasons. This explains the derivation of the **cha** command. In the first form *filename* must not conflict with any VICOM or installed *vsh* command; in the second form this restriction does not apply. If *filename* already contains an extension (i.e. there is a '.' in the name) the extension '.vc' is not applied. *Vsh* first searches in the current working directory for a macro file and then in a global directory defined by *vsh*, called the *vsh* macro directory. Your working directory will contain macros which you have developed for your own use, whereas the *vsh* macro directory contains macros of more general use, some of which are described in this document. You may list the contents of the *vsh* macro directory, and read the files in that directory, in order to learn of macros not described in this document. In order to print the path name of the *vsh* macro directory, issue the installed command

•> pvd

Examples of macros are shown in Figures 5.1 and 5.2. These macro files, **sobel.vc** and **snap.vc** can be invoked as in the following examples:

•> cha sobel 1 2 3 4
•> sobel 2 3 1 1
•> snap 1
•> cha snap.vc 4

5.2 Parameters

Parameters can be passed to a macro file, or to *vsh*, or to an invocation of *vsh* (see section 5.4). Parameters are given on the command line. Thus the command

•> cha *filename p1 p2 ... p9*

invokes the macro *filename* with the parameters *p1 p2 ... p9* which will be used for substitution in the macro.

Parameters are words (sequences of Ascii characters) separated by spaces, commas, or other punctuation (except '_', '-', '.', and '•' which are treated as letters). Parameters can contain embedded spaces, commas or other delimiters provided the entire parameter is enclosed in single or double quotes (which are not part of the parameter). There can be up to 9 parameters on a command line.

The parameter values for *p1*, *p2*, ..., *p9* are substituted for the symbols *%1*, *%2*, ..., *%9* respectively, in the macro. Thus every occurrence of the character sequence *%1* is replaced entirely by the complete word specified by *p1* in the **cha** command. In general, *%n* is replaced by

```
* sobel A B C D
*       detects edges using the Sobel operator.
*
*       A       source image.
*       B,C     work images.
*       D       destination image.
*
*       Note: A must be distinct from B & C
*
mas %1>%2 (13)
mag %2>%2
mas %1>%3 (15)
mag %3>%3
add %2,%3>%4
lsh %4>%4 (-3)
```

Figure 5.1. Sample chain file, **sobel.vc**.

```
* snap A
*       Takes a snapshot
*
*       A       Image to receive the picture.
int
dis %1
cam %1
. Say cheeeese, then press ^D.
dig %1
two %1>%1
res
```

Figure 5.2. Chain file to take a snapshot, **snap.vc**.

the n^{th} parameter, where n = 1, 2, ..., 9. Within the macro the symbol *%0* (percent zero) is replaced by the name of the macro file (actually the first word of the command line excluding **cha**).

The parameters (eg. *%1*) may appear in any location of a macro file. The symbol is completely replaced by the entire character sequence of the corresponding argument. The parameters can represent image numbers, constants, file names, keywords, portions of file names, image file names, programs, parameters to programs, parameters to other macros, etc.

Figures 5.1 and 5.2 show macro files with parameters. The lines beginning with '*' are comments and are ignored by *vsh*. All symbols of the form *%n*, where *n*, is a digit, will be replaced

by the parameters in the call to the macro file.

5.3 Defaulting Parameter Values

If a parameter is not defined when a macro (or *vsh* level) is invoked then the corresponding symbol (*%n* where *n* = 1, 2, ..., 9) is generally replaced by the empty string when it is encountered in the macro. Default values for missing parameters can be specified within a macro via the **default** statement. The form is

•> default *p1 p2 ... p9*

The delimiters between parameter values are exactly as described for a macro invocation. The parameters are specified in the same order as in the macro invocating statement and empty defaults can be specified just as empty arguments are specified (see Section 5.2). Multiple **default** statements within a macro are allowed, the most recent default value for a particular is parameter is always used.

The parameter value corresponding to *%n* is deemed to be missing if there are fewer than *n* parameters in the invocation, or if the n^{th} parameter is the empty string delimited by commas or other non-blank delimiters. For example the third parameter (%3) is missing from the following invocations:

•> macrofile parm1 parm2
•> macfile2 (p1, p2) (p4, p5, p6)

5.4 Nesting of Macro Calls

Macro file calls can be nested. There is a restriction on the depth of nesting tied to the number of open files or descriptors that a process may have at one time.

Conceptually, it is convenient to say that *vsh* is always processing a macro file. When *vsh* is invoked by the shell, the current 'macro file' is usually the standard input (which in turn is usually the user's terminal). *Vsh* processes this input essentially the same as from any macro file. When the user invokes a macro file, a new level of *vsh* is created with its own source of input. This input is processed until an end of input is encountered, at which time control is returned to the higher level which continues processing from its input source. It is possible from within a macro file to invoke a level of *vsh* that gets its input from the user's terminal (or more correctly the top level standard input); this is done via the pause commands described below. We will use the interchangeable terms, "macro file level", "source input level", or simply "*vsh* level" to refer to this nesting of *vsh* invocations.

The important points here are that generally, what is true for macro file invocations is true at any *vsh* level and that one level of *vsh* must complete before the invoking level continues. When the input to *vsh* is a terminal there are some differences. One is that *vsh* prompts for input on terminals and the other is that errors are reported to source terminals as human readable messages and do not cause the *vsh* level to terminate.

5.4.1 Pause Commands. The pause command has three forms:

. *message.*
- *message.*
pau *message.*

where *message* stands for any character string. A pause command first displays *message* and then invokes a new *vsh* level with the standard input as the source. The primary function of the pause commands is to allow interactive control during the processing of a macro file. Processing returns to the invoking level when the invoked level of *vsh* is terminated with a ^D, **end**, or **quit** command

(see Section 5.5).

In a user's program the statement

i = vsh("-")

invokes *vsh* with the standard input (usually the user's terminal) as the source. The user is then free to perform *vsh* commands. When the user terminates the *vsh* level control is returned to the user's program.

Figure 5.2 contains an sample macro file, **snap.vc**, that will digitize an image into a given image buffer. **Snap.vc** first sets up the VICOM so that the camera image is displayed in the desired image. Then the pause command

is executed, and the prompt "Say cheese, then press ^D" is displayed on the VICOM user's terminal, followed by a prompt like

•••>

At this point the scene should be prepared. The user can use any *vsh* command, including invoking macro files. When the scene is prepared the user terminates the pause *vsh* level with a ^D, **end**, or **quit** command. **Snap.vc** then continues with the **dig** command.

When the source input to a *vsh* level is a terminal *vsh* prompts for the input. The number of asterisks in the prompt indicates the *vsh* level. The top level prompt for *vsh* is '•>' at deeper levels there are more asterisks, for example '•••>' at the third level. The purpose is to remind the user of the depth of nesting.

5.4.2 Scope of Parameters. The parameters passed to a *vsh* level are available only to that level. If a new level is invoked the parameters current at the time of invocation are pushed onto a stack and are replaced by new parameters. When control returns to the original level its parameters are restored from the stack.

Parameter substitution is the first action performed on the command line. It is performed on the line as a whole without respect to word boundaries or command line syntax. Substitution may be performed in the keyword of a command. Arguments may be concatenated and the result is that the corresponding parameter symbols are concatenated. Parameter values may be passed to deeper levels by including the symbols which are substituted by the parameters among the parameter list invoking the next level, as in

•> cha *macro* %3 %2

5.5 Macro File Termination

A nesting level is terminated, and control is passed to the level immediately above, upon execution of an **end**, **quit**, or **EOF** command. A source level is also terminated by the end of the source file or a ^D on the terminal. Of course, termination of the top level *vsh* terminates *vsh* entirely. When *vsh* is invoked from a user's program, for example

i = vsh(*command*)

it terminates when the *command* is complete. If the command invokes a macro file (or is a pause command) *vsh* terminates when the macro file terminates.

5.6 Comments

Any command with a '*' or '#' in the first column is a comment and is not parsed or executed. If the command line begins with '**' the line is a comment which will always be displayed on the user's terminal. This is true even if **set notrace** is active (see Section 5.8.1). This allows the user to provide a commentary during the execution of a macro file.

5.7 Do command

The **do** command is a simple control command which enables repeated execution of a command. The format is

*>do *n command*

where *n* is the number of times that the command *command* is to be repeated. *Command* can be any *vsh* command (including **do**). The **do** will be prematurely terminated if an error occurs during the execution of *command*.

5.8 Set Command

The **set** command allows *vsh* processing options to be set or reset. There are two options described in this section, they are trace and step.

5.8.1 Tracing Macro Files. It is sometimes desirable to have the commands of a macro file displayed as they are performed. This can be done by issuing the following command

*> set trace

This causes all commands at this level and deeper levels to be displayed on the VICOM users' terminal while they are processed. The displayed commands are preceded on the display by

filename>

where *filename* is the name of the macro file in which the command is contained.

The trace can be turned off by

*> set notrace

5.8.2 Single Step Processing. In the single step mode *vsh* displays each line of a macro file before executing it and waits for input from the user's terminal. If an empty line is entered (i.e. just a <return>), *vsh* executes the command and goes to the next command, which is displayed and so on. Any other input line is treated as a *vsh* command and is executed and *vsh* again waits for input. This continues with the original macro file command suspended until an empty line is entered. Once entered, the single step mode applies to the current macro level and deeper levels, except at levels which receive input from a terminal. Single step mode can be entered and exited via the **step** command as follows:

*> set step
*> set nostep

5.8.3 Asynchronous Operation. In normal operation *vsh* issues commands to the VICOM and then waits for the VICOM's response. The command

```
*> set async
```

puts *vsh* in the asynchronous mode. Commands are issued and responses received in an asynchronous manner. This speeds up the processing of macro files, but as a result the direct correlation between the commands and their responses (in particular errors) is lost. The asynchronous mode also effects the use of *vsh* from users' programs. In asynchronous mode a call to **vsh** or **vicom** will normally return immediately indicating no error, however an error condition resulting from the current command may be reported upon return from a later call to one of the routines. For this reason the asynchronous mode should be avoided until a program is throughly debugged. In addition certain interactive commands, for example **pcr** and **roa**, always operate in synchronous mode. The synchronous mode of operation can be restored by the command

```
*> set sync
```

5.9 Errors and Interrupts

When an error occurs in a *vsh* or VICOM command that the user has entered on the terminal, *vsh* responds with an error message that should indicate the problem.

When an error occurs during the execution of a macro file with input from a source other than a terminal, the macro file is terminated and the error is passed to the *vsh* level that invoked the macro. This level will also terminate if its input is not from a terminal and so on until the error reaches a *vsh* level that has a terminal for input. At this point the appropriate error message is displayed. In short, an error during the processing of a macro file causes *vsh* to return control to the nearest level receiving commands from a terminal.

If the user sends an interrupt signal to the *vsh* process (usually done by typing ^C), an error is forced in the current *vsh* command. Thus an interrupt causes *vsh* to return to the nearest level receiving commands from a terminal. When *vsh* is waiting for an instruction from a terminal an interrupt will cause the current level to terminate with an error, thus causing control to return to the next level with a terminal as input.

The stop signal (usually sent by typing ^Z) causes an asynchronous pause. The effect is that *vsh* will invoke a new level of *vsh* with the standard input as the source at the next convenient point (usually the completion of the current instruction). The user can then enter *vsh* commands. When the new *vsh* level is terminated control returns to the previous level at the point it was interrupted. This provides a way of temporarily stopping execution of a macro file, interactively inserting *vsh* commands, and resuming execution at the place where the macro was suspended.

6 Shell Commands

Executable programs can be invoked from *vsh*. The formats are

•> prog p1 p2 ... p9
•> **ex** *prog p1 p2 ... p9*

where *prog* is the name of the program and *p1 p2 ... p9* are parameters to be passed to the program. The keyword **ex** is optional; however, if it is not used, *prog* cannot conflict with any VICOM or *vsh* command or any of the available macro files. The available macro files include the macro files in the current working directory and those in the global *vsh* macro file directory. The **ex** form is also preferable because it is slightly faster. If the **ex** keyword is missing *vsh* must search the directories for available macro files before determining that *prog* is an executable program. The **ex** keyword immediately signifies an executable program and thus eliminates the search.

7 Image I/O

Image data can be stored in VICOM image memories, files on VAX storage media, and arrays or buffers in the user's program. This section describes how the user can transfer image data between the various media.

A VAX file is usually a file on the disk, either in the user's current working directory, or the *vsh* image directory. However, a VAX file might also be a tape file, or located on some other device. *Vsh* allows one to transfer data between the VICOM image memories and VAX files by using *vsh* commands. These commands are described in Section 7.1.

Procedures exist to facilitate the transfer of image data between program arrays or buffers and external media such as VAX files or VICOM image memories. These procedures are available in C, FORTRAN, and Pascal and are described in Section 7.2.

To communicate with VICOM image memory and VICOM look-up-tables, the procedures use names for VICOM buffers which make them look like UNIX special files. The names are listed in figure 7.1. Sections 7.1 and 8 describe the use of these special file names. The number of frame buffers available depends on the configuration of the VICOM. See the *VICOM Users' Manual* for details of the VICOM frame buffers and look-up-tables.

Image transfers involve moving a lot of data across the VAX unibus, and should be considered an expensive operation. Further, a 512 by 512 by 8 bit image occupies a quarter megabyte. Since disk storage space is always at a premium, users should never store entire images on disk for long periods of time (hours).

7.1 *Vsh* Commands For Image Transfers

This section describes the *vsh* commands and macros for transferring images between the VICOM and VAX files.

File Name	*Description*
/dev/vicom1	512 x 512 x 16 image memory 1
/dev/vicom2	512 x 512 x 16 image memory 2
/dev/vicom3	512 x 512 x 16 image memory 3
.	
.	
.	
/dev/vicom16	512 x 512 x 16 image memory 16 (if available)
/dev/vicompp	4096 x 16 point processor look-up table
/dev/vicomr	1024 x 8 red display look-up table
/dev/vicomg	1024 x 8 green display look-up table
/dev/vicomb	1024 x 8 blue display look-up table

Figure 7.1. VICOM Related Special Files.

7.1.1 Load and Store. The most common format for image storage is one pixel per 8 bit byte. The easiest way to transfer images in this formats between the VICOM image memory and VAX files is with the **load** and **store** commands. The store command is used to store a VICOM image memory on a VAX file, and load is used to move a VAX image file onto the VICOM. The forms are:

•> load *A filename*
•> store *A filename*

Here *A* is the image file number, and *filename* is the VAX file name, with a default extension of '.img'. That is, if no extension is present in the character string *filename*, then the extension '.img' is appended to the file name before processing. Thus almost all VAX image files will have the extension '.img'. It is also assumed that the VICOM is configured for 512 by 512 images (i.e. the default configuration of the **mem** command). The file used by **store** will be the in the current working directory, if a complete pathname is not specified. On the other hand, **load** causes *vsh* to search for the image file in the current working directory, and then in the *vsh* image directory, if a full path is not specified. The location of the *vsh* image directory can be determined from the *vsh* command "•> pvd".

For example,

•> load 1 oilcan
•> store 3 ../mom

loads the 512 by 512 image 1 from the file oilcan.img, and stores image 3 on the file ../mom.img.

Both **load** and **store** are in fact macro commands, and are equivalent to the commands

•> rim *A* (*filename*, 2, 8196)
•> wim *A* (*filename*, 2, 8196)

respectively, as described in the next section.

7.1.2 Rim and Wim. The **rim** command is used to read an image from a VAX file to the VICOM. Conversely, the **wim** command is used to write an image from the VICOM to the VAX. Note that the commands are named from the perspective of the VICOM (which is confusing, since the commands are submitted to *vsh*, which is operating on the VAX). The formats are

•>rim *A* (*filename*, *mode*, *blocksize*)
•>wim *A* (*filename*, *mode*, *blocksize*)

where *A* is the image number (1, 2, ...) on the VICOM. It is generally assumed that the VICOM is in the 512 by 512 image mode (i.e., a default **mem** command has been issued). The parameter *filename* is the name of a VAX file, with a default extension of '.img'. That is, if no extension is present in *filename*, then the extension '.img' is appended. The *mode* parameter determines whether or not the VAX file is packed. *Mode* = 1 means that the data is transferred 16 bits per pixel, whereas *mode* = 2 means 8 bits per pixel. The *mode* parameter may be omitted if the *blocksize* parameter is also omitted, in which case *mode* = 2 is assumed. *Blocksize* is the number of pixels per block for the transfer. This must be a power of 2 less than or equal to 8192. Generally, the larger the *blocksize*, the faster the transfer. *Blocksize* = 8192 is assumed if the parameter is omitted.

The syntax described for **wim** and **rim** is as defined by VICOM Systems Inc. in the VICOM manual. However, the parenthesis and comma delimiters can be replaced with spaces.

For the **rim** command, if *filename* is not a complete path name, then *vsh* searches for the VAX file first in the user's current working directory, and then in a *vsh*-defined directory of images available for general use. This directory is called the *vsh* image directory. You can print the path name of the *vsh* image directory by issuing the *vsh* command

```
•> pvd
```

(print *vsh* directories). You can then list and read images in that directory.

The **wim** command always writes (or overwrites an existing file) in the current working directory, if a complete pathname is not specified.

If *mode* = 1, then only 12 of the 16 bits per pixel transferred by the **wim** command from the VICOM to the VAX have significance. The low order 4 bits will be undefined. That is, the graphics data is not transferred by **wim**. In order to transfer 16 bit images from the VICOM to a VAX file, you should first issue a **store**, followed by a **swa** (swap bytes), followed by another **store** to a different file, and another **swa** to restore the image data back to the original format. On the other hand, *mode* = 1 transfers by **rim** from VAX files to the VICOM successfully transfer 16 bit data, and load graphics data if the graphics nibble is not write protected.

Examples of **wim** and **rim** are:

```
•> wim 2 (oilcan)
•> rim 1 (envoy, 1, 4096)
```

The first stores image 2 in a file 'oilcan.img', using one byte per pixel. The **rim** command loads image 1 from a file called 'envoy.img', using 2 bytes per pixel, and transfer in blocks of 4096 pixels.

The **rim** and **wim** commands behave a bit peculiarly if the VICOM is not in the standard 512 by 512 image mode (i.e., if a **mem** command has been issued with nonstandard image size). If the images do not have 512 columns (i.e., '•>mem(*r*, *c*)' where *c* is not 512) the result of a transfer will be predictable but undoubtedly not what is desired. If, however, the images have 512 columns but not 512 rows, the appropriate image number will be transferred correctly, and a image file will contain something other than a quarter million pixels. Thus partial VICOM image memories can be accessed by **mem**'ing to 256 by 512, 128 by 512, etc.

7.1.3 Rdm and Wdm. The *vsh* commands **rdm** and **wdm** are used to transfer partial images on the VICOM, using one block for the transfer. The **rdm** command transfers a single block of data in a VAX file to the start of a designated VICOM image memory. The **wdm** command transfers from the start of a VICOM image memory one block of data to a VAX file. Although it is not possible to access an interior portion of a VICOM image memory in a fixed memory configuration, as with **rim** and **wim**, it is possible to **mem** down to a smaller image with nonstandard number of rows and 512 columns, and then use **rdm** or **wdm** to access the relevant subimage.

The form of the commands is:

```
•> rdm A (filename, mode, blocksize)
•> wdm A (filename, mode, blocksize)
```

The parameters are the same as in **rim** and **wim**. The only difference is that **rdm** and **wdm** transfer exactly one block, whereas **rim** and **wim** transfer a sufficient number of blocks so that the entire image is transferred.

7.2 Image I/O Involving Program Variables

User programs can access image data in VAX files and in VICOM image memories. The same subroutines are used to access both VAX files and VICOM memories. Image data can be read or written, using subroutines available in C, Fortran, or Pascal. The routines are in the *vsh* library. When compiling a program using these routines use the '-lv' option to access the library. By image files, we mean data that usually represents image intensity or feature values, organized by rows and columns. However, image files can also be used to store histogram vectors and table data for look-up-tables. The procedures described below can be used to access these generalized image files.

Image I/O using these routines is a four step process. First, the image file must be opened. VAX files are named with an absolute or relative pathname; VICOM memories are named by the device names (see Figure 7.1). Next, the user must declare any nonstandard attributes of the image file. Attributes determine how the data in the image file corresponds to image values. Default attributes are applied if none are declared. Third, data is read or written using the appropriate procedure. Different procedures are provided for different types of array variables (floating point, integer, etc.). Finally, the image file should be closed.

One of the attributes of an image is its pixel data type: images can be fixed point, unsigned integer, signed integer, floating, or complex. Reads and writes also have types, and refer to the variable type of the array containing the pixel data. If the types match, then no conversion is needed in assigning data. If the types differ, then an implicit conversion takes place during the read or write.

7.2.1 Opening an Image. To open an image for reading or writing from C, Fortran, or Pascal, use the function procedure:

imd = iopen(*file*, *rwmode*)

Here *file* is a character string or constant giving the VAX file or VICOM memory to be opened, and *rwmode* is 0 for read, 1 for write, and 2 for read and write access. For VAX files, *file* is simply the path name of the file; for VICOM memories, *file* is of the form '/dev/vicomx', where *x* = 1, 2, ..., pp, r, g, or b. The returned value is an integer value which will be used to designate the image file in the remaining image I/O procedures. The image descriptor imd and the mode rwmode variables are of type int, Integer, and int in C, Fortran, and Pascal respectively.

The VICOM point processor and VICOM display tables can be accessed only in write mode (*rwmode* = 1). Certain protected VAX disk image files can only be opened in read mode (*rwmode* = 0).

7.2.2 Declaring Attributes. The attributes that an image file can have are listed in Figure 7.2. Not all attribute values are legal for all image media. VICOM memories have a restricted range of attribute values. Certain attribute values restrict the range of other attributes.

The *type* attribute determines how the bit string values in each pixel are translated into pixel values. The *access* attribute is either sequential or direct; sequential access implies that the image will be accessed left to right, top to bottom, successive pixels, starting in the first row, first column. Direct access image files can be read in any order. In the image file, each pixel can have 1, 2, 3, ... or up to 64 bits of significance. The number of bits used to store this information is usually the same as the number of bits of significance, but can be more. These two parameters are specified by the attributes *nb* and *nstore* respectively. The *hilo* attribute specifies whether the *nb* bits come from the high or low part of the *nstore* storage bits. If the image *type* is 'fixed point' then *hilo* defaults to the high order portion of the storage, otherwise *hilo* defaults to the low order portion. If the image file has *access* attribute 'direct', then it is possible to read or write a subsampled subimage of the image file. This virtual image will have *nrows* rows and *ncols* (after subsampling), starting in the *frow* row and *fcol* column, choosing every *xsamp* pixel along each row and every *ysamp* pixel along each column. Here, *nrows*, *ncols*, *xsamp*, *ysamp*, *frow*, and *fcol* are positive integer valued attributes. When writing a subimage, the remaining pixels will remain either unchanged or undefined. If *ysamp* $\neq$ 1 or *frow* $\neq$ 1 or if *ncols*$*$*xsamp* is not the full width of the image file, then the width of the actual image must be specified in the *width* attribute. When reading or writing a subimage, the pixels of the subimage are numbered left to right, top to bottom, within the subimage, starting with pixel number 1 at location (*frow*, *fcol*) and ending with pixel number *nrows*$*$*ncols* at the lower right corner of the subimage.

Attribute values are generally defaulted. The default values depend on whether the image file is a VAX disk, VICOM image memory, VICOM point processor memory table, or VICOM

Attribute	Reference # (FORTRAN)	Reference Name (C & Pascal)	Allowed Values VAX file	Allowed Values VICOM frame	Allowed Values VICOM pp	Allowed Values VICOM display
type	1	*type*	1=fixed point 2=unsigned int 3=signed int 4=floating 5=complex	1 2 3	1 2 3	1 2 3
access	2	*access*	1=direct 2=sequential (Tape only sequential)	1, 2	2	2
significant bits	3	*nb*	1,2,...,64	8,12,16	16	8
storage bits	4	*nstore*	*nb*,...,64	8,12,16	16	8
high or low order bits	5	*hilo*	1=high 2=low	1,2	1,2	1,2
number of rows	6	*nrows*	1,2,...	1,..,512	4	1
number of columns	7	*ncols*	1,2,...	512	512	512
x-sampling	8	*xsamp*	1,2,...,*width*	1	1	1
y-sampling	9	*ysamp*	1,2,...	1	1	1
first row	10	*frow*	1,2,...,*width*	1	1	1
first column	11	*fcol*	1,2,...	1	1	
image width	12	*width*	1,2,...	512	512	512

Figure 7.2. Image Attributes.

display table. Attribute values can be changed at any time the image file is open. Attribute values that are not specifically declared will retain default values, based on the image type and the current declared and defaulted attribute values. The procedure for changing attribute values is:

i = iattr(*imd*, *iattrs*)

where *i* is an integer returned error code (see Appendix C), and *imd* the image descriptor designating the image file, and *iattrs* is a pointer to the structure of attribute values. In C, *iattrs* is a structure which is declared by the lines:

```
#include <iattr.h>
struct iattributes *iattrs;
```

Likewise in Pascal, *iattrs* is a pointer to a record structure defined by the lines:

```
#include <iattrpas.h>
var iattrs: iattrpointer;
```

In Fortran 77, *iattrs* is an integer array containing 20 values:

Integer iattrs(20)

The elements of the attribute are addressed by their names in C such as *iattrs->type* and *iattrs->access* and in Pascal *iattrs^.type* and *iattrs^.access*. In Fortran, the elements are accessed by number: *iattrs*(1) for the *type*, and *iattrs*(2) for the *access* attribute and so on. Names and numbers, as well as allowed values and meanings for those values are given in Figure 7.2.

Attribute values are declared by calling **iattr** with the appropriate positive integer values in attribute elements to be declared, and zero values in attribute elements to which default values are to be applied. A call to **iattr** with zero values in any attribute elements will cause default values to be applied to those attributes, regardless of the previous attribute values. Before **iattr** is called, default values are applied to all attributes. Thereafter, attribute values many be changed any

Attribute		VAX file	VICOM frame	VICOM pp	VICOM display
1. *type*		1	1	1	1
2. *access*		1 (tape=2)	1	2	2
3. *nb*					
	type=1	12	12	16	8
	type=2	8	12	16	8
	type=3	8	12	16	8
	type=4	32	———	not allowed	———
	type=5	64	———	not allowed	———
4. *nstore*		*nb*	*nb*	*nb*	*nb*
5. *hilo*		———	1 if *type* = 1, otherwise 2		———
6. *nrows*		512	512	4	1
7. *ncols*		512	512	512	512
8. *xsamp*		1	1	1	1
9. *ysamp*		1	1	1	1
10. *frow*		1	1	1	1
11. *fcol*		1	1	1	1
12. *width*		———	*ncols***xsamp*		———

Figure 7.3. Default Attribute Values.

number of times. The default attribute values are described in Figure 7.3.

Attribute values for an image are not stored with the image. It is the responsibility of the user's program to know the attributes of a previously created image. It is possible to declare different attributes to an image than those with which is was created. Sometimes, this is a useful feature.

7.2.3 Reading, Writing, and Seeking. The procedures for reading and writing data in image files have the form:

ireadx	fixed point 1	unsigned integer 2	signed integer 3	floating point 4	complex 5
ireadi	$x*2^{nb-1}$ $nb \le 32$	x $nb \le 31$	x $nb \le 32$	int(x) $nb=32$	int(modulus(x)) $nb=64$
ireads	$x*2^{nb-1}$ $nb \le 16$	x $nb \le 15$	x $nb \le 16$	int(x) $nb=32$	int(modulus(x)) $nb=64$
ireadf	x $nb=32$	float(x)	float(x)	x $nb=32$	modulus(x) $nb=64$
ireadz	complex(x,0.)	complex(float(x),0.)		complex(x,0.)	x $nb=64$
ireadb	———	true if x≠0, false if x=0			———
ireadc	x $nb=8$	x $nb=8$	x $nb=8$	not allowed	not allowed

x	hi/low order *nb* bits of image file datum depending on the *hilo* attribute. Interpreted according to the *type* of the image.
int()	convert argument to an integer.
float()	convert argument to a 32 bit floating point.
complex()	convert arguments to a 64 bit complex number.
modulus()	modulus of the complex argument.

Figure 7.4. ireadx Conversions.

num = ireadx(*imd*, *buf*, *n*)
num = iwritx(*imd*, *buf*, *n*)

where *x* = i (integer)
s (short integer)
f (floating)
z (complex)
b (boolean)
c (character).

Thus the procedures are **ireadi**, **ireads**, **ireadf**, **ireadz**, **ireadb**, and **ireadc** for reading and **iwriti**, **iwrits**, **iwritf**, **iwritz**, **iwritb**, and **iwritc** for writing. For *x* = i, s, f, z, b, c, the array *buf* is of type Integer (32 bit integer), short integer (16 bit integer), floating (i.e., real), complex, boolean, or character respectively. When reading, data is transferred from the image file to the array variable *buf*, one pixel per array element, subject to conversion according to the image file type (as declare in the image file attributes) and the read type, as indicated by *x* = i, s, f, z, b, or c. The manner of the conversion is described in Figure 7.4. For writing, data is transferred from the array variable to the image file, one element per pixel, subject to the conversion as described in Figure 7.5. Figure 7.4 gives the interpretation of the value loaded into an element of *buf*, given the *nb* bits of significance from the image file. The figure also gives restrictions on *nb* in order for a read to be legal. In Figure 7.5, the bits of the elements of *buf* are used, and are translated and loaded into *nb* bits of the *nstore* bits used to represent the pixel. Generally, the remaining bits are zero filled. However, when writing an integer, short integer, or character value into an image of type signed integer, the low order bits are loaded with the 2's complement value, and the nb^{th} bit is sign extended to the left to fill *nstore* bits. There are some restrictions on *nb* noted. Further, in order for the results to be meaningful, the values of the *buf* array should lie within certain ranges. These ranges are noted.

In ireadx and iwritx, the transfer of the *n* pixels of data begins with the current pixel, and transfers successive pixels. Pixels are numbered left to right, top to bottom, in the virtual (subsampled) image file described by the attributes. The current pixel is 1 when the image is opened, and is generally one greater than the last pixel number read or written. The current pixel number can be changed, however, if the file has 'direct' access attribute, by using the procedure **vseek**. The form is:

i = vseek(*imd*, *offset*, *origin*)

where *imd* is the integer image descriptor designating the image file, *offset* is the integer pixel number to become the current pixel number, measured relative to the location specified by *origin*. If the integer value *origin* = 0, then *offset* is the pixel number relative from the start of the image. If *origin* = 1, then *offset* measures ahead from the current pixel location, and if *origin* = 2, then *offset* measures back from the last pixel in the image. The integer *i* is an error code, described in Appendix C.

7.2.4 Closing an Image File. Image files that have been opened by **iopen** are closed by normal termination of the program, or explicitly by calling:

i = iclose(*imd*)

where *imd* is the integer image descriptor and *i* is an integer returned error code (see Appendix C). It is sometimes necessary to close image files because not too many files can be open at one time. To rewind a sequential access image file, it suffices to close it and then to reopen it. Image files that are closed by **iclose** or by normal termination of the program will be saved by the file system.

iwritx	fixed point 1	unsigned integer 2	signed integer 3	floating point 4	complex 5
iwriti or iwrits	$b/2^{nb-1}$ $-2^{nb-1} \le b < 2^{nb-1}$ $nb \le 32$	b $0 \le b < 2^{nb}$ $nb \le 31$	b $-2^{nb-1} \le b < 2^{nb-1}$ $nb \le 32$ (sign extended)	float(b) $nb=32$	complex(b,0.) $nb=64$
iwritf	int($b*2^{nb-1}$) $-1.0 \le b < 1.0$ $nb \le 32$	not allowed	not allowed	b $nb=32$	complex(b,0.) $nb=64$
iwritz	not allowed	not allowed	not allowed	not allowed	b $nb=64$
iwritb	if b true 1 otherwise 0	1 0	1 0	1. 0.	complex(1.,0.) complex(0.,0.)
iwritc	b $nb=8$	b $nb=8$	b $nb=8$ (sign extended)	not allowed	not allowed

b	value of an element of the array *buf*. Interpreted as specified by x in vwritx.
int()	convert argument to an integer.
float()	convert argument to a 32 bit floating point.
complex()	convert arguments to a 64 bit complex number.

NOTE: Values in the table are converted to the type specified by the *type* attribute, and written to the high or low order bits of the *nstore* storage bits, according to the *hilo* attribute.

Figure 7.5. Data values written by iwritx.

7.2.5 Examples. Some examples follow. Suppose we open an image file for reading, with attribute *type* = 'fixed'. Then reading using **ireadf** will produce values in the array *buf* of reals in the range $-1. \le buf(i) < 1.$ An integer read **ireadi** will produce integer *buf* values in the range $-2^{(nb-1)} \le buf(i) \le 2^{(nb-1)}-1$. A short integer read **ireads** will produce the same values providing $nb \le 15$. However, if the image is declared to have 'unsigned integer' type, then **ireadi** (and **ireads**, providing $nb \le 15$) will return values in the range $0 \le buf(i) \le 2^{nb}-1$.

Suppose we open a disk image file for writing, and declare it to be of *type* 'fixed', with nb = 4 (four bits of significance per pixel). To save storage space, we allow the attribute *nstore* to default to *nstore* = 4. We may then write using **iwritf**, loading *buf* values in the range $-1. \le$

buf(i) < 1., noting that since there are only 4 bits of precision, there are actually only 16 quantization levels. We can use integer *buf* values in the range −8, ..., 7, by making use of **iwriti**, or short integer *buf* values in the same range by making use of **iwrits**. If we insist on thinking of the values as lying in the range 0, 1, ..., 15, then we should declare the image to have type attribute 'unsigned integer', and then load *buf* values in the range 0, ..., 15 using **iwriti**.

The above example gives a disk file with 4 bits per pixel, and cannot be loaded onto the VICOM with the load or **rim** *vsh* commands. Suppose we wish to write a disk file which can later be loaded. Then the image file should have *nb*=8, *nstore*=8, *ncols* = *nrows* = 512, *xsamp* = *ysamp* = *fcol* = *frow* = 1, and be of *type* fixed, unsigned integer, or signed integer. (Default attributes suffice!) If the *type* is 'fixed', (which is the default), then data can be written using **iwritf** and floating *buf* values in the range [−1, 1), or by using **iwriti** and integer *buf* values in the range −128, −127, ..., 127. To consider the data in the range 0, 1, ..., 255, then the *type* attribute should be declared 'unsigned integer', and **iwriti** or **iwrits** used with nonnegative *buf* values. Regardless of whether the image was written sequentially or direct access, after the image file is closed, it can be written to a VICOM memory by the **load** *vsh* command (see Section 7.1).

The **rim** command can be used (with *mode* = 1) to load 16 bit data onto the VICOM. To write a disk file which can later be **rim**'ed to the VICOM with *mode* = 1, or to write 16 bits directly to the VICOM, open the image file with attributes *nb* = 16, *nstore* = 16, *nrows*, *ncols*, *xsamp*, *ysamp*, *fcol*, and *frow* defaulted, and *type* either 'fixed', 'unsigned integer', or 'signed integer'. Graphics data will be loaded onto the VICOM in the low order 4 bits of the 16 bits transferred. With unsigned integer *type*, **iwriti** should be used with *buf* values in the range 0, 1, ..., $2^{16}-1$. To load the upper 12 bits, leaving the graphics nibble unchanged, the data VICOM memory should be write protected. In this case, it is often more convenient to write the image with *nb* = 12, *nstore* = 16, and *type* = 'fixed' (with *hilo* = 'hi'). In this case **iwriti** takes *buf* values in the range −2048, ..., 2047, and loads them into the high order 12 bits. Keeping the same attributes, but setting *type* = 'unsigned integer', then **writi** takes *buf* values in the range 0, ..., 4095.

A disk image which has been created by the **wim** *vsh* command, using *mode* = 1 transfer, contains only 12 bits of precision in each 16 bit pixel. This disk file can be read by opening the image file, and declaring attributes of *nb* = 12, *nstore* = 16, and the remaining attributes defaulted. Then **ireadf** yields floating *buf* values in the range [−1, 1), and **ireadi** will yield integer *buf* values in the range −2048, ..., 2047. If integer values in the range 0, 1, ..., 4095 are desired, then the image can be declared to have *type* = 'unsigned integer' (instead of 'fixed') and *hilo* = 'hi', and then read using **ireadi** or **ireads**. Note that a **ireadf** in the latter case would yield floating *buf* values in the range 0., 1., ..., 4095.

Of course, a disk image file created by a **wim** transfer using *mode* = 2, or equivalently, a **store** command, can be read using default attributes and **ireadf** to yield values in the range [−1, 1), or **ireadi** or **ireads** to yield values in the range −128, ..., 127. To obtain values in the range 0, ..., 255, declare *type* = 'unsigned integer', *nb* = 8, and use **ireadi** or **ireads**. In the latter case, **ireadc** can also be used to obtain the data in character form, which in C can also be used in arithmetic expressions.

Although the VICOM memories should be accessed sequentially, the data on the disk may be read or written in direct access mode (i.e., **vseek** may be used). Moreover, an image file that has 16 bits per pixel, or any other size, may be declared with *nb* = 8, and *type* 'fixed' (say), and then read using **ireadc** or **ireadi** to obtain raw bit data about each byte in each pixel. In this case, sequential read traverses the image low order bytes to high order bytes, left to right, top to bottom.

8 Look-Up-Tables

The VICOM contains four look-up-tables. The point processor is a 12 bit in − 16 bit out look-up-table, and is the primary table for performing point operations. There are three color display look-up-tables − one each for the red, green, and blue display channels. These tables can be used for point operations, but are generally used purely for pseudo color display. The display look-up-tables are 10 bit in − 8 bit out.

Many VICOM commands can be used to load predefined look-up-tables into the point processor. For example, **exp**, **log**, **lin**, **thr**, and **sli** are commands that load look-up-tables, and then perform the look-up operation. These commands can also load the same tables into the display look-up-tables. See the section of Appendix B on point operations for a complete list.

The VICOM command **poi** can be used to perform a look-up operation if the correct data is already loaded into the point processor look-up-table.

For look-up-tables that cannot be constructed from any of the predefined VICOM point operations, mechanisms are provided which allow the user to load user-defined look-up-tables. There are two general methods for accomplishing this. First, one can write a program which uses one of the subroutines defined in Section 8.1 below, such that when the program is executed a table is created and loaded into the appropriate VICOM look-up-table. The second possibility is to load a precompiled table. Precompiled tables are created by the same subroutines but the table is stored in a disk file instead of being loaded directly to the look-up-table. This precompiled table can then be loaded into the appropriate VICOM look-up-table with *vsh* commands (described in Section 8.2). The advantage to precompiled tables is that they can be reloaded as often as desired without the expense of recalculating the table values. The first option, calculating the table on-line, is useful if the table is to be used very rarely or when the table changes dynamically.

The routines described below are intended to shield the user from the mechanics of loading VICOM look-up-tables. However, the user must be aware that loading any of the look-up-tables from user defined tables (as opposed to built-in VICOM commands) overwrites the first 4096 pixels in the first of the 512 by 512 VICOM images (i.e. the first 8 rows of the image). Of course, if the VICOM is **mem**'ed to non-standard configuration the affected pixels may spread over several images.[2]

For completeness, the data representations of the VICOM look-up-tables is described in Section 8.3.

8.1 Loading Tables

Two routines are provided for loading look-up-tables. The first, *wrlut*, requires an externally defined floating-point function. The second, *wrluti*, uses an integer array. To include these routines in a program the user must use the option '-lv' on the compile or load statement for the program using them (see Section 3.5).

Wrlut has the following form:

wrlut(*tablename*, *func*);	for C or Pascal
call wrlut(*tablename*, *func*)	for FORTRAN

where *tablename* is a character string and *func* is an externally defined function. *Tablename* tells *wrlut* where to put the table as shown in Figure 8.1. *Func* must be a floating-point function with a single floating-point argument. The function must be defined on the interval

$$-1 \le x < 1,$$

and the range of the function is the same interval,

[2]The reason for this unfortunate situation is that the FOS VICOM can only exchange data with the host computer via the image storage areas.

$$-1 \leq func(x) < 1$$

Note that if loaded into one of the color display table, $func(x) = 0$ will result in no intensity for that color, $func(x)$ near to 1 or $func(x)$ near -1 results in the midrange intensity for that color, and $func(x) = -1/128$ results in full intensity.

It is sometimes useful to define look-up-tables in terms of the unsigned integers or bit strings. The procedure wrluti allows the user to create and store an *n*-bit in, *m*-bit out look-up-table. The form is

wrluti(*tablename*, *itab*, *n*, *m*);	for C or Pascal
call wrluti(*tablename*, *itab*, *n*, *m*)	FORTRAN

Tablename is a string that is interpreted just as for *wrlut*. *Itab* is an array (or in C a pointer to an array) which contains 2^n integer values. The arguments *n* and *m* are integers.

The arguments *n* and *m* are the number of input and output bits of the look-up-table, respectively. The restrictions on *n* and *m* are, $1 \leq n \leq 12$ and $1 \leq m \leq 16$. When the look-up-table is applied, the high order *n* bits of the image data is converted into an index, *i*, into the array *itab*, where *itab[0]* is the first element of the array. The low order *m* bits of the value *itab[i]* becomes the high order *m* bits of the output. Thus there must be 2^n entrys in the array *itab* and each entry should satisfy

$$0 \leq itab[i] < 2^m \qquad \text{where}$$
$$0 \leq i \leq 2^n - 1$$

The conversion of an image datum into an index is straightforward. For the point processor, the high order *n* bits are converted into an unsigned integer in the range 0 to 2^n-1, and becomes the index. For display look-up-tables, the high order 10 bits of the pixel value is padded with two low order 1 bits, and then the high order *n* bits of the resulting 12 bit value is made into the index.

If $m < 16$, then all unused low order output bits will be manifest as 0. If the table is loaded into a display look-up-table, and $m > 8$, then only the high order 8 bits of the low order *m* bits of *itab[i]* are used in the output.

tablename =	/dev/vicompp	point processor look-up-table
	/dev/vicomr	red display look-up-table
	/dev/vicomg	green display look-up-table
	/dev/vicomb	blue display look-up-table
	filename	file *filename*.tbl if no extension is specified.

Figure 8.1. Look-up-table names.

8.2 Loading Precompiled Look-up-tables

If *wrlut* or *wrluti* has been used to create a table on a disk file, then that table can be loaded into the VICOM's point processor with the following *vsh* command

•> ldpp *tablename*

Where *tablename* is the string used when the table was created. A more general *vsh* command

•> ldlut *tablename lut*

loads the display look-up-tables or the point processor look-up-table. *Tablename* is as before. The *lut* parameter is a value from 0 to 4 indicating the look-up-table to be loaded, according to:

lut	Look-up-table
0	point processor
1	red display
2	green display
3	blue display
4	red, green, and blue displays

Note that the VICOM command **pol** must be used to perform the look-up operation after the point processor table has been loaded.

8.3 VICOM Data Structures for Look-up-tables

This section describes, in some detail, the VICOM's use of look-up-tables. It is not necessary to read this section to use the VICOM's look-up-tables.

The VICOM command **luk** is used to transfer data from the first 4096 16-bit values in the image memory into the point processor look-up-table, or one of the display look-up-tables. Let us interpret those 4096 values as 16-bit unsigned integers in the range 0 to $2^{16}-1$, and denote the table as T[0], T[1], ..., T[4095]. The files created by *wrlut* and *wrluti* contain 4096 16-bit values, and are identical to the 16-bit image data T[0], ..., T[4095], as in a disk image file (see Section 7.1).

The data is loaded from the image memory into the point processor look-up-table by the VICOM command

•> luk (0) or
•> luk

The look-up-table operation can be applied by the VICOM command **pol**. During that operation, the high order 12 bits of each input pixel value is interpreted as an unsigned integer in the range 0 to 4095 to yield an index *i*. The output of the look-up operation is the 16 bit T[*i*], which is written in the output pixel. If the four graphics bits of the output pixel is write-protected, then only the high order 12 bits of T[*i*] are used.

The data is loaded into one or more display look-up-tables by the commands

•> luk (1) loads the look-up-table for the red display
•> luk (2) loads the look-up-table for the green display
•> luk (3) loads the look-up-table for the blue display
•> luk (4) loads the look-up-tables for the all three displays

For display tables, 1024 values are used, namely, T[3], T[7], T[11], ..., T[4095]. The display output value is determined from the high order 10 bits of the pixel intensity value. These 10 bits are padded with two one bits, and the resulting 12-bit value is interpreted as an unsigned integer in the range 0 to 4095 to yield an index *i*. The output display look-up-table will be the high order 8 bits of the 16-bit value T[*i*].

Appendix A

This section lists the VICOM commands in alphabetical order. The syntax for each command is followed by a brief description followed by a list of VICOM processors involved in the command. The abbreviations for the processors are: PP for the point processor, M for the micro processing unit, DC for the display controller, VC for the video controller, and AP for the array processor. For descriptions of the processors see "VICOM User's Guide".

VICOM Commands

Command	*Description*	*Processor*
ADD A,B > C (a)	a=1 scaled, 2 unscaled	PP
ADK A > B (a, b)	a=const, b=1 scaled, 2 unscaled	PP
AND A,B > C	logical AND of images	PP
ANK A > B (const)	logical AND with const	PP
ARE A (xc,yc,plane)	area within closed contour	M
ASH A > B (n)	n=# of bits to the right, sign extended	PP
BAR A	bar chart	M
BIT (bit,dc)	bit slicing, bit=1-8(msb to lsb)	DC
BOX A (xc,yc,h,w,val,back)	box of val with background level back	M PP
CAM A (a,b)	a=1 interlace, 2 non-interlace	VC DC
	b=1 internal control, 2 external	
CDM (count)	continues DMA	IO
CHE A (x,y,n,v)	checks image data starting at (x, y)	M
	vs. the n values in v	
CHK A (x1,y1,x2,y2,const,n)	check image block vs. const n exceptions listed	M
COL A,B,C	display color	DC
CON A (const)	constant image	PP
COP A > B	copy	PP
CUR A (x,y,plane)	position cursor at (x, y)	M
DEF (a)	a=0 no display after process, 1=display	DC
DEV (a,b)	a=0 device disable, 1 enable	M
	b=1 tablet, 2 joystick, 3 ball	
DIG A (a)	a=1 bottom byte, 2 top byte digitization	VC DC
DIK A > B (const)	division of const*0.1 by image	PP AP
DIL A > B,X,Y (x1,y1,x2,y2)	expand image block to full size	M AP PP
DIN A,Z (a)	frame digit and temp intergration	VD DC PP
	a=number of frames must be power of 2	
DIS A	display A	DC
DIV A,B > C	0.1*A/B (clipped at ± 1.)	AP PP
DOT A (x1,y1,sh,sw,h,w,dot,back)	dot chart	M PP
EDG A > B,Z (a)	a=1 prewitt sqrt, 2 prewitt mag,	AP PP
	3 sobel sqrt, 4 sobel mag	
ELL A (xc,yc,major,minor,ang,f,plane)	ellipse f=1 draw, 0 erase	M
END	only with VERSADOS	M
ERF A > B (g,dc)	g=0 guassian error function, 1 inverse	PP DC
EXP A > B (a,dc)	a=1-4 exponential	PP DC
EXT A	print max and min pixel values	M
FAL A,B,C (a)	false color 6 perms of RGB	DC
FIL A (xc,yc,plane)	fill closed contour	M
FLI A,B (n)	flicker with period of n frames	DC

VICOM Commands

Command	*Description*	*Processor*
GRA A > B (plane,dc)	B(i,j)=T[A(i,j)]	PP DC
	T obtained from graphics plot	
GRY A (a)	a=1 horiz, 2 vert, 3 diag	M
HEQ A > B (dc)	histogram equalization	M PP DC
HIP A > B,Z (n,const)	high pass filter n=iterations const=0-9	PP AP
HIS A (low,high,plane)	histogram plotted in graphics	M
IMP A (x,y)	maximum impulse at x,y	PP
INT (dc)	tables for integer mode	DC
INV A > B (dc)	0.1/A (clipped at ±1.)	PP DC
LEN A (x1,y1,plane,outline,rs,cs)	length of minimally connected line	M
LIN A > B (clipl,cliph,oclipl,ocliph,dc)	linear point transformation	PP DC
LMP (dc)	load DC from origin to cursor	M
LOG A > B (a,dc)	log function, a=1-4 (inv to EXP)	PP DC
LOO (t,n)	loop through images 1-n, display each t frame times	DC
LOP A > B,Z (n,const)	low pass filter n=iterations const=0-9	PP AP
LSH A > B (n)	logical shift n bits right	PP
LUK (dc)	load look-up from image memory from image 1	M PP DC
LUT (dc)	load look-up from microcomputer memory (DOS)	M PP DC
MAG A > B (dc)	absolute value	PP DC
MAS A > B (a,v,b)	a=0 user defined 3x3 linear mask in v	AP
	1 North 2 NE 3 E 4 SE 5 S 6 SW 7 W 8 NW	
	9-11 Lapl 12-13 horz 14-15 vert 16-18 lopass	
	19-21 hipass 22 v.line 23 h.line 24 0.5 diag	
MED A > B (v,h)	plus shaped median filter v by h (1,3,5,7)	M
MEM (rows,cols)	configures logical memory	M
MER A,B,C > D (plane)	merge pair under control of mask	PP
MOM A (a)	a=1 mean, 2 mean variance skewness	M
MOV (filen,dwell,n,reels,loopn)	movie presentation (DOS)	DC DISK MP
MUK A > B (const)	multiply by const	AP
MUL A,B > C	C = A*B	MP AP
NEG A > B	B = -A	PP
NOT A > B	logical negation	PP
ONE A	constant image FFFFH	PP
ORK A > B (const)	logical OR with const	PP
ORR A,B > C	logical OR of images	PP
OUT A > B (threshold)	remove outliers in a neighborhood	M
OVR (color1,color2,color3,color4)	overlay graphics colors (1-8)	DC
PCR	prints pixel cursor location and amplitude	M
PER A (xc,yc,plane,outline,rs,cs)	perimeter of closed contour	M
PLO A (pa,pc,dc,d)	plot lookup table in graphics	M DC
	using pa for axis & pc for curve dc=0 PP	M DC
	c=2 DC green c=3 DC blue d=1 full scale d=2 +ve quad	
POI A > B	B(i,j)=table[A(i,j)] using PP table	PP
POW A > B (a,dc)	a=1 cube root, 2 sqrt, 3 square, 4 cube	PP DC
PRI A (x1,y1,c)	prints c by 8 pixel block on console	M
PSE (a)	pseudocolor	DC
PUT A (x1,y1,h,w,amp)	writes block of constant value	M
QUI	terminates vicom command operations(DOS)	M

VICOM Commands

Command	*Description*	*Processor*
RAN A (a)	random image a=seed must be odd	M
RDM A (a,number 16bit words)	read image block a=1 16bit, 2 8bit	IO
REA A (filename,a)	read from disk a=1 16bit, 0 8bit (DOS)	DiskC
REC A (xc,yc,h,w,ang,f,plane)	rectangle f=1 draw, 0 erase	M
RED A > B,X,Y (x1,y1,x2,y2)	reduction into block	AP M
REP	repeat chain file sequence (DOS)	M
RES (cam,pse,roam,scr,zoo)	reset to default (0 disable, 1 enable)	M
RIO A (a,number of rows)	read image a=1 16bit, 2 8bit	IO
ROA (xc,yc)	image roam	DC
ROT A > B,X,Y (xc,yc,degs)	rotation about (xc, yc)	AP M
RUB A > B (clipl,cliph,oclipl,ocliph,x,y,dc)	rubber band x,y inflection pts.	PP DC
SAM A > B (x1,y1,h,w,vsamp,hsamp,x2,y2)	subsampling	M
SCA A > B (fract cliph,fract clipl,dc)	linearly scale image to range [0, 1]	PP DC
SCR A (dx,dy)	image scroll using offset (dx, dy)	DC
SDM (a)	suspend active DMA a=0 status, 1 stop DMA	IO
SLI A > B (low,high,bckgrnd,dc)	slicing, bckgrnd=1 zero, 2 image	PP DC
SPL A,B > C (a)	split screen a=1 lA,rB; 2 lA,lB; 3 rA,rB	DC
SUB A,B > C (a)	a=1 scaled, 2 unscaled	PP
SUK A > B (const,a)	subtract const, a=1 scaled, 2 unscaled	PP
SVD A > B,Y,Z (a)	convolution a=file of 3x3 kernels (DOS)	AP PP
SWA A > B	reverses top and bottom bytes	PP
TEX A (x1,y1,ang,f,e,plane,'text')	write text f=0 small, 1 large font	M
THR A > B (thresh,const,bckgrnd,dc)	threshold, above thresh set to const rest set to bckgrnd	PP DC
TRA (plane)	trace cursor	M
TRH A > B,Z (vshift,hshift)	image translation	PP AP
TRI A,B,C > D (a,b,c)	tricolor transform D=aA+bB+cC	AP
TRL A > B (vshift,hshift,c)	translation c=unmapped amp.	M PP
TRP A > B	transpose image	M
TWO A>B	integer to two's complement	PP
UNS A > B,Z (a,low pass impulse response)	unsharp mask a=proportion	PP AP
VEC A (x1,y1,x2,y2,e,plane)	line e=1 draw, 0 erase	M
WAI	delay execution for 0.5 secs (DOS)	M
WAL A > B,Z (a,b,c,d,e,f)	wallis stats, mean, std dev, normalization	AP PP
WDM A (a,number 16bit words)	write image block a=1 16bit, 2 8bit	IO
WIO A (a,number of rows)	write image a=1 16bit, 2 8bit (DOS)	IO
WRI A (file,a)	write to disk a=1 16bit, 2 8bit (DOS)	DiskC
WRP (a)	0 write protect/enable 1 not protected/disable 2 write protect/disable 3 not protected/enable	M
XFR (file)	4096 words transferred to micro buffer (DOS)	M IO
XOK A > B (const)	xor with const	PP
XOR A,B > C	exclusive OR of images	PP
ZER A	all pixels zero	PP
ZGR A (plane)	zero graphic memory plane=0 all	PP
ZOO (a)	zoom by factor 1-6	DC

Appendix B

This section lists the VICOM commands grouped by operation categories. The processor abbreviations are the same as for Appendix A.

ALU Operations

Command	*Description*	*Processor*
ADD A,B > C (a)	a=1 scaled, 2 unscaled	PP
ADK A > B (a, b)	a=const, b=1 scaled, 2 unscaled	PP
AND A,B > C	logical AND of images	PP
ANK A > B (const)	logical AND with const	PP
ASH A > B (n)	n=# of bits to the right, sign extended	PP
DIK A > B (const)	division of const*0.1 by image	PP AP
DIV A,B > C	0.1*A/B (clipped at ± 1.)	AP PP
LSH A > B (n)	logical shift n bits right	PP
MUK A > B (const)	multiply by const	AP
MUL A,B > C	C = A*B	MP AP
NEG A > B	B = -A	PP
NOT A > B	logical negation	PP
ORK A > B (const)	logical OR with const	PP
ORR A,B > C	logical OR of images	PP
SUB A,B > C (a)	a=1 scaled, 2 unscaled	PP
SUK A > B (const,a)	subtract const, a=1 scaled, 2 unscaled	PP
XOK A > B (const)	xor with const	PP
XOR A,B > C	exclusive OR of images	PP

Display Operations

Command	*Description*	*Processor*
COL A,B,C	display color	DC
DIS A	display A	DC
FLI A,B (n)	flicker with period of n frames	DC
LOO (t,n)	loop through images 1-n, display each t frame times	DC
MOV (filen,dwell,n,reels,loopn)	movie presentation (DOS)	DC DISK MP
ROA (xc,yc)	image roam	DC
SCR A (dx,dy)	image scroll using offset (dx, dy)	DC
SPL A,B > C (a)	split screen a=1 lA,rB; 2 lA,lB; 3 rA,rB	DC
ZOO (a)	zoom by factor 1-6	DC

Analyze Operations

Command	*Description*	*Processor*
EXT A	print max and min pixel values	M
HIS A (low,high,plane)	histogram plotted in graphics	M
MOM A (a)	a=1 mean, 2 mean variance skewness	M

Graphics Operations

Command	*Description*	*Processor*
ARE A (xc,yc,plane)	area within closed contour	M
ELL A (xc,yc,major,minor,ang,f,plane)	ellipse f=1 draw, 0 erase	M
FIL A (xc,yc,plane)	fill closed contour	M
LEN A (x1,y1,plane,outline,rs,cs)	length of minimally connected line	M
OVR (color1,color2,color3,color4)	overlay graphics colors (1-8)	DC
PER A (xc,yc,plane,outline,rs,cs)	perimeter of closed contour	M
PLO A (pa,pc,dc,d)	plot lookup table in graphics	M DC
	using pa for axis & pc for curve dc=0 PP	M DC
	c=2 DC green c=3 DC blue d=1 full scale d=2 +ve quad	
REC A (xc,yc,h,w,ang,f,plane)	rectangle f=1 draw, 0 erase	M
TEX A (x1,y1,ang,f,e,plane,'text')	write text f=0 small, 1 large font	M
VEC A (x1,y1,x2,y2,e,plane)	line e=1 draw, 0 erase	M
ZGR A (plane)	zero graphic memory plane=0 all	PP

I/O Operations

Command	*Description*	*Processor*
CDM (count)	continues DMA	IO
RDM A (a,number 16bit words)	read image block a=1 16bit, 2 8bit	IO
RIO A (a,number of rows)	read image a=1 16bit, 2 8bit	IO
SDM (a)	suspend active DMA a=0 status, 1 stop DMA	IO
WDM A (a,number 16bit words)	write image block a=1 16bit, 2 8bit	IO
WIO A (a,number of rows)	write image a=1 16bit, 2 8bit (DOS)	IO
XFR (file)	4096 words transferred to micro buffer (DOS)	M IO

Neighborhood Operations

Command	*Description*	*Processor*
EDG A > B,Z (a)	a=1 prewitt sqrt, 2 prewitt mag,	AP PP
	3 sobel sqrt, 4 sobel mag	
MAS A > B (a,v,b)	a=0 user defined 3x3 linear mask in v	AP
	1 North 2 NE 3 E 4 SE 5 S 6 SW 7 W 8 NW	
	9-11 Lapl 12-13 horz 14-15 vert 16-18 lopass	
	19-21 hipass 22 v.line 23 h.line 24 0.5 diag	
MED A > B (v,h)	plus shaped median filter v by h (1,3,5,7)	M
OUT A > B (threshold)	remove outliers in a neighborhood	M
UNS A > B,Z (a,low pass impulse response)	unsharp mask a=proportion	PP AP
WAL A > B,Z (a,b,c,d,e,f)	wallis stats, mean, std dev, normalization	AP PP

Cursor Operations

Command	*Description*	*Processor*
CUR A (x,y,plane)	position cursor at (x, y)	M
DEV (a,b)	a=0 device disable, 1 enable b=1 tablet, 2 joystick, 3 ball	M
LMP (dc)	load DC from origin to cursor	M
PCR	prints pixel cursor location and amplitude	M
TRA (plane)	trace cursor	M

Filter Operations

Command	*Description*	*Processor*
HIP A > B,Z (n,const)	high pass filter n=iterations const=0-9	PP AP
LOP A > B,Z (n,const)	low pass filter n=iterations const=0-9	PP AP
SVD A > B,Y,Z (a)	convolution a=file of 3x3 kernels (DOS)	AP PP

Generate Operations

Command	*Description*	*Processor*
BAR A	bar chart	M
BOX A (xc,yc,h,w,val,back)	box of val with background level back	M PP
CON A (const)	constant image	PP
DOT A (x1,y1,sh,sw,h,w,dot,back)	dot chart	M PP
GRY A (a)	a=1 horiz, 2 vert, 3 diag	M
IMP A (x,y)	maximum impulse at x,y	PP
ONE A	constant image FFFFH	PP
RAN A (a)	random image a=seed must be odd	M
ZER A	all pixels zero	PP

Geometric Operations

Command	*Description*	*Processor*
DIL A > B,X,Y (x1,y1,x2,y2)	expand image block to full size	M AP PP
MER A,B,C > D (plane)	merge pair under control of mask	PP
RED A > B,X,Y (x1,y1,x2,y2)	reduction into block	AP M
ROT A > B,X,Y (xc,yc,degs)	rotation about (xc, yc)	AP M
SAM A > B (x1,y1,h,w,vsamp,hsamp,x2,y2)	subsampling	M
TRH A > B,Z (vshift,hshift)	image translation	PP AP
TRL A > B (vshift,hshift,c)	translation c=unmapped amp.	M PP
TRP A > B	transpose image	M

Spectral Operations

Command	*Description*	*Processor*
FAL A,B,C (a)	false color 6 perms of RGB	DC
PSE (a)	pseudocolor	DC
TRI A,B,C > D (a,b,c)	tricolor transform D=aA+bB+cC	AP

Video Operations

Command	*Description*	*Processor*
CAM A (a,b)	a=1 interlace, 2 non-interlace b=1 internal control, 2 external	VC DC
DIG A (a)	a=1 bottom byte, 2 top byte digitization	VC DC
DIN A,Z (a)	frame digit and temp intergration a=number of frames must be power of 2	VD DC PP

Point Operations

Command	*Description*	*Processor*
BIT (bit,dc)	bit slicing, bit=1-8(msb to lsb)	DC
ERF A > B (g,dc)	g=0 guassian error function, 1 inverse	PP DC
EXP A > B (a,dc)	a=1-4 exponential	PP DC
GRA A > B (plane,dc)	B(i,j)=T[A(i,j)] T obtained from graphics plot	PP DC
HEQ A > B (dc)	histogram equalization	M PP DC
INT (dc)	tables for integer mode	DC
INV A > B (dc)	0.1/A (clipped at ±1.)	PP DC
LIN A > B (clipl,cliph,oclipl,ocliph,dc)	linear point transformation	PP DC
LOG A > B (a,dc)	log function, a=1-4 (inv to EXP)	PP DC
MAG A > B (dc)	absolute value	PP DC
POI A > B	B(i,j)=table[A(i,j)] in PP table	PP
POW A > B (a,dc)	a=1 cube root, 2 sqrt, 3 square, 4 cube	PP DC
RUB A > B (clipl,cliph,oclipl,ocliph,x,y,dc)	rubber band x,y inflection pts.	PP DC
SCA A > B (fract cliph,fract clipl,dc)	linearly scale image to range [0, 1]	PP DC
SLI A > B (low,high,bckgrnd,dc)	slicing, bckgrnd=1 zero, 2 image	PP DC
THR A > B (thresh,const,bckgrnd,dc)	threshold, above thresh set to const rest set to bckgrnd	PP DC

Utility Operations

Command	*Description*	*Processor*
CHE A (x,y,n,v)	checks image data starting at (x, y) vs. the n values in v	M
CHK A (x1,y1,x2,y2,const,n)	check image block vs. const n exceptions listed	M
COP A > B	copy	PP
DEF (a)	a=0 no display after process, 1=display	DC
END	only with VERSADOS	M
LUK (dc)	load look-up from image memory from image 1	M PP DC
LUT (dc)	load look-up from microcomputer memory (DOS)	M PP DC
MEM (rows,cols)	configures logical memory	M
PRI A (x1,y1,c)	prints c by 8 pixel block on console	M
PUT A (x1,y1,h,w,amp)	writes out block	M
QUI	terminates vicom command operations(DOS)	M
REA A (filename,a)	read from disk a=1 16bit, 0 8bit (DOS)	DiskC
REP	repeat chain file sequence (DOS)	M
RES (cam,pse,roam,scr,zoo)	reset to default (0 disable, 1 enable)	M
SWA A > B	reverses top and bottom bytes	PP
TWO A>B	integer to two's complement	PP
WAI	delay execution for 0.5 secs (DOS)	M
WRI A (file,a)	write to disk a=1 16bit, 2 8bit (DOS)	DiskC
WRP (a)	0 write protect/enable 1 not protected/disable 2 write protect/disable 3 write protect/enable	M

Appendix C

By convention any *vsh* function that returns a return or error code returns zero in normal circumstances, a non-zero returned value indicates some sort of exceptional condition (not always serious). Some routines return codes that indicate more precisely the exceptional condition, the returned codes are listed in this appendix. The first list contains errors from within *vsh*, the codes and names are defined in the header file **vsh.h**. The second list contains error returned from the VICOM, many of the errors are not meaningful for our system. The *vsh* function **verrdesc**, given an error code, returns a pointer to the description of the error. The following code fragment performs a *vsh* command and then prints the error message if any.

```
#include <vsh.h>

int  rc;
char *command;

rc = vsh (command);
if (rc != NO_ERROR)
     printf (" sh error -- %s , verrdesc(rc));
```

Vsh Related Error Codes

Code	*Name*	*Description*
0	NO_ERROR	No error.
1	ABORT	Chain file aborted.
2	NO_CHILD	No child process in vex, disaster.
3	END_OF_FILE	EOF or end encountered.
4	CHAIN_OVERFLOW	Too many chain files.
5	NO_COUNT	No count on 'do' statement.
6	CANT_DO	Can't do this command.
7	CANT_OPEN	Can't open image file.
8	BAD_IO	Disk I/O error during image transfer.
9	DMA_DISABLED	Attempted transfer while DMA disabled.
10	BAD_PARAMETER	Bad parameter on *vsh* command.
11	NO_FLAG	No such flag.
12	EX_ERROR	Error in shell command.
13	GOT_INTR	Received interrupt signal.
14	WHATISTHIS	What was that?
15	DMA_LOCKED	DMA locked by some other processes.
16	NO_RESPONSE	No response from VICOM to command.

VICOM Error Codes

Code	*Description*
64	Required device not available.
65	Command is ambiguous, use longer form.
66	Command is unknown, try another.
67	An operand is missing from the command.
68	An invalid image number is specified.
69	Command format is incorrect.
70	An invalid count is specified.
71	An invalid numeric constant is specified.
72	An invalid file name is given.
73	Unable to assign the specified file.
74	I/O error occurred while transferring image.
75	Disk error occurred while closing file.
76	Unable to allocate a file for image.
77	Error in loading lookup table.
78	Device is currently busy.
79	Incompatible images have been specified.
80	Graphics commands are disabled.
81	Cursor not active.
82	Command is not yet implemented.
83	Command not valid for this memory configuration.
84	Device write protected.
85	Error detected during check.
86	Warning - text string truncated.
87	Outside edge of contour not found.
88	Unexpected error from system call.
89	Device still active.
90	Illegal MOVIE file specified.
91	Error while assigning volume.
92	Movie loop out of sync with display controller.

Appendix D

This is a list of the known problems with *vsh* to be used in conjunction with the "*vsh* Users' Manual". It includes bugs as well as unimplemented features.

(1) The **pvd** command is not implemented.

(2) The VICOM's frame buffers and look-up-tables are not yet available as special files. They can be accessed by the routines described in Section 7 of the *vsh* manual.

(3) Parameters can not contain embedded delimiters as described in Section 5.2. At present a parameter to a macro file must be a word surrounded by delimiters. The quotes, both single and double, are treated as delimiters. Also, making an empty parameter between commas etc. does not work as desired.

(4) Default parameter values as described in Section 5.3 do not yet exist. An undefined parameter is left UNCHANGED.

(5) The implementation of interrupts does not quite match the description in section 5.9. When invoked by the shell command, *vsh*, ^C will cause a chain file to abort an return to the closest level receiving input from a terminal. Pressing ^Z will normally cause the current process to stop returning control to the shell.

(6) The ".tbl" extension described in Section 8.1 is not yet implemented. Look-up-tables have the same extension as images, ".img". The point processor table can be loaded with the *vsh* command, **table**.

NYU CS TR-113 c.1
Clark, Dayton

VSH user's guide: a
software environment for